Nicholas Tomlinson

Network Anomaly Discovery

Computer Science Tripos

Robinson College

May 16, 2012

Proforma

Name	**Nicholas Tomlinson**
College	**Robinson College**
Project Title	**Network Anomaly Discovery**
Examination	Computer Science Tripos Part II, 2012
Word Count[1]	11918
Project Originator	Nicholas Tomlinson
Supervisors	David Evans and Malcolm Scott

Original Aims of the Project

My project will discover anomalous behaviour of a computer network by analysis of traffic reaching an end machine, and by sensibly selected generation of traffic from this machine (for example, traceroutes). This would be done by constructing a framework with which specific anomalies could be recognised, and other functionality provided. It should be noted that, since the project is intended to be run on an end machine rather than elsewhere in the network, anomalies centre largely around the machine in question. For example, the detection of port scanning would typically be a port scan against the particular machine running the program.

Work Completed

I have implemented a modular framework and set of modules for performing real time analysis of traffic on a computer network. The system runs on a typical end user machine and is able to identify several different network anomalies. This work has been completed with all of the basic functionality specified despite the very open ended nature of the problem, and the large number of ways in which it could be designed. The modularity of the project does, of course, mean there is great scope to take the project much further.

Special Difficulties

None

[1] Estimated using **catdvi counted.dvi | wc -w** – counted.dvi includes chapters 1 through 5 only.

Declaration of Originality

I Nicholas Tomlinson of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

This dissertation describes a project designed to discover anomalous behaviour of a computer network[1]. In many cases, this anomalous behaviour may be malicious – such as described below, but this may not always be the case. As observed by my project supervisor, Dr David Evans, this project is at its most general an attempt to discover information about a computer network from the incomplete information available to an end machine. It is this attempt to discover information from an unprivileged position on the network (but with administrative access to the local machine) in an unobtrusive way that makes this project interesting.

Note here: "an unprivileged position"! As discussed in Section 1.2, intrusion detection systems (IDSs) already exist that are designed to run in the core of a network – but few systems exist that are designed to detect malicious operation of the network from the perspective of a normal user's machine, and even fewer that are prepared to use active techniques to investigate the network. It is also important to note that not all anomalies are outright malicious, although clearly some are. They may be as innocuous as control traffic that was not anticipated, or simply be the result of incorrect configuration. Indeed, many anomalies have been discovered during the development of the system that were entirely benign.

Towards the end of the production of this dissertation, Virgin Media (who provide my home Internet connection) were ordered by a UK court to block access to The Pirate Bay. This served as a good real world example of my project in action. This interception was detected first time without any modification to the system. Appendix E demonstrates this.

This project has successfully implemented a system to perform passive and active analysis of network traffic, and is able to establish the presence

[1] The definition of an anomaly in the context of this project anomaly is given in Section 2.1.1.

of several anomalies. This dissertation concerns itself with how this was accomplished, and why it was accomplished in this way.

1.1 History

I first became interested in the detection of network anomalies in December 2008 when the Internet Watch Foundation (IWF) attempted to block the Wikipedia page on the Scorpions album "Virgin Killer" in what proved to be a rather conspicuous way[2][21][19]. I have made a number of interesting observations about the system employed by Virgin Media, both during this original incident, and subsequently[3]. Several UK ISPs, including Virgin Media, redirect traffic in a way similar to a system known as CleanFeed. CleanFeed operates by intercepting specific IP addresses for filtering with an HTTP proxy[18]. The HTTP proxy is then able to block specific URLs rather than entire domain names or IP addresses. This is, of course, only one example of anomalous network behaviour that I would like to be able to detect.

1.2 Previous Work

There has been much work in the area of IDSs such as Snort[15] that are run by network administrators on nodes within the network, as well as on firewalls that are designed to run on end user systems. IDSs are intended to permit the network administrator (but often not users of the network, such as home ISP users) to monitor the network for traffic that may indicate a security issue, or use of the network outside the organization's use policy. Firewalls are predominantly intended to detect and block attacks (such as a port scan) targeted at the specific machine, and provide security by reducing the number of services visible to the network. In general, such firewalls are passive and make little or no attempt to identify traffic that does not represent a direct

[2] In particular, the majority of Wikipedia traffic from the UK originated from one of a small number of HTTP proxies, forcing Wikipedia's administrators to disable anonymous editing from these IP addresses, and as a result from the majority of UK users. It is interesting to analyse the precise way in which this attempt at censorship was achieved, as it appears quite poorly implemented in many cases.

[3] Some high profile sites with user submitted content, for example one click file sharing systems, often seem to appear in the IWF's list. This became evident upon analysis of the data I gathered in an attempt to use machine learning for HTTP fingerprinting. It has also previously been observed that (at the time of the observation) at least 25% of the IWF's list of redirected sites are legitimate file hosting sites[20].

attack on the local machine. They will not, for example, highlight evidence of traffic that has been intercepted, as described in Section 1.1. Traditional IDSs that run within the network provide little reassurance to the user that does not trust the competence or intentions of the network administrator. In cases where an IDS is run on the user's end system, it will typically also be passive.

1.3 General Relation to Computer Science

In addition to the previous work discussed in Section 1.2, this project has relations to many areas of Computer Science and the Computer Science Tripos – most notably Computer Networking, Security I, Principles of Communication, and Security II. In addition to these topics, the Software Engineering, Object Oriented Programming, and Software Design courses are relevant to any project of this size.

Chapter 2

Preparation

This chapter of the dissertation discusses what had to be learned before implementation of the project could begin. It should be noted that, while some of the design could be appropriately placed in this chapter, this is discussed in Chapter 3.

2.1 Requirements Analysis

This section discusses what the project should be able to do upon completion. The most obvious objective is to be able to identify a number of different anomalous behaviours that may be present on a computer network. There is a huge variety of different anomalies that one could seek to detect; it is clearly not practical to attempt to implement them all in a Part II project. Therefore I will aim to implement a selection of suitable anomalies to demonstrate the working of my project. The project should be designed in a modular way – allowing additional anomalies (and other functionality to be discussed later) to be easily added. The project should be capable of running on a machine with limited resources – such as a typical laptop of a few years ago, and without generating unacceptable traffic.

2.1.1 Definition of an Anomaly

In the context of this project, an anomaly is any unexpected observable behaviour of a computer network to which the end system is attached. Typically, this will be unusual or non-compliant behaviour or traffic, however user specification may also dictate that traffic that would otherwise be considered normal is in fact anomalous, or that traffic that would normally be considered anomalous is in fact normal. An anomaly may be likely to be ma-

licious (such as a port scan, or traffic interception), however it may also be unintentional behaviour (such as the Ethernet frame with a six byte payload that is occasionally produced by my phone, I believe erroneously) or just a curiosity (such as a change in network configuration). Section 2.1.2 provides examples of anomalies that could be detected.

2.1.2 Possible Identifiable Anomalies

This section discusses what types of network traffic we might wish to detect, and how we might go about doing so – from a fairly high level point of view. Let us consider how we might detect some anomalies[1].

Unusual Path to Destination

If our traffic is being intercepted, it is possible that the behaviour of the network will reflect the interception if the path through the network is changed as a result. This might show up in a traceroute[2]. In this case, we could examine a traceroute for evidence of interception, such as:

- The presence of a node (or subnet) we know to be involved in interception.
- A significant difference (such as a detour) in the traceroute of addresses we would not expect a difference to appear in. In this case, we must be careful because the network may be legitimately configured to do this.

Unusual Hop Time

If a packet takes an unusually long time to reach a node in the network, it may be the result of congestion or poor network configuration. This is an example of a non-malicious anomaly. It may be revealed by timing the responses of a traceroute, and highlighting any round trip time that is over a specified threshold.

Port Scanning

A port scan is an attempt to determine whether any of a set of ports are open or not – typically to find services running on the machine that could potentially be vulnerable to exploitation[23]. The set of ports probed by

[1] In fact, we discuss here the anomalies suggested in the project proposal.

[2] Indeed I have observed that several ISPs exhibit this behaviour. An explanation of tracerouting is given in Section 3.4.10.

a port scan will often be far longer than could be reasonably explained by accidental causes such as entering the wrong port or IP address. This case can be detected by keeping a record of access attempts. Any machine that attempts to access more than a set threshold of ports may be considered to have performed a port scan.

Unusual Packet Types

Computers communicating over a computer network generally use packets that consist of several protocols organised in a layered fashion. For example, a TCP segment might be nested in an IPv4 packet, which might in turn be nested in an Ethernet frame. Figure 2.1 demonstrates this. In the case of Ethernet and IPv4, protocols are assigned numbers[3][5]. This makes the task of identifying packet types trivial: compare the packet's protocol number against a list of known numbers. Many higher layers rely on the application to determine the protocol (often by conventional port number on the server, but this cannot be relied upon[14]). Different packet types may therefore be identified. If we have a list of packet types that are usual, any packet arriving that is not of one of those types may be considered to be unusual. Most obviously, the system could be provided with an explicit list of protocols that are expected. A more advanced implementation could perform statistical analysis of previous traffic to identify protocols that have not previously been encountered.

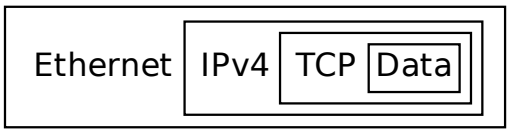

Figure 2.1: An example of a TCP packet

2.2 Development Process

2.2.1 Development Model

In line with good software engineering principles, I plan to adopt a development model that is suitable to a project of this nature, and the resources available to me. The model chosen should have scope for a single person to experiment with the system and change it in response to difficulties or ideas that are discovered during development. There should also be scope for good software engineering practice such as thorough testing, and clear specification of the requirements. I have chosen the evolutionary development model

principally to accommodate experimentation and continual testing of small increments – it is much easier to test a small change to a system that is known to otherwise work than it is to leave most of the testing to the end as is typical in the waterfall model. The specification can be gradually worked towards and potentially amended slightly as new ideas or difficulties are encountered.

2.2.2 Design Paradigms and Choice of Language

It is helpful to break the project down into small manageable chunks; therefore, I intend to produce a highly modular design. To assist in this, I intend to make use of object oriented programming. This project lends itself well to the production of modules that are isolated from each other – except for modules that have clear interaction (such as the IPv4 module that acquires its data from the Ethernet module). Modules that interact with each other should do so in a way that is consistent throughout the system, and that makes this interaction easy to understand in the code. It will be very helpful to devise a way to manage the complexity of how the modules are linked together, however this is a discussion for Chapter 3.

This project does a lot of relatively low level networking tasks, and must be able to handle a lot of data efficiently – for example, without copying it unnecessarily. It is also desirable (though not strictly necessary) that the project be portable across different operating systems and hardware. In order to support the design paradigms specified in this section, I require that the chosen language support object oriented programming to reduce the incidental complexity, but with support for relatively low level operations. With these criteria in mind, I have chosen C++ as the primary language of this project.

2.2.3 Testing

Testing is an extremely important part of the development process, so it pays to give some consideration to how it will be performed. I favour an evolutionary development process – where features are added and tested.

Since I will be using the evolutionary development model, it is important to ensure testing occurs throughout the development process – not just of new parts of the system, but also of existing parts. This way, changes to the system that might conflict with parts that have already been implemented are discovered quickly. Fortunately, this project lends itself well to continual testing.

I also plan to develop a number of testing tools – mechanisms to provide simulated network traffic, tools to evaluate the resource requirements of the

project, and so on. These tools can be run on a regular basis to check for regressions that occur as the result of subsequent development.

2.3 Preparatory Study

This section will detail the various tools, software libraries, documentation, and so on that had to be learned and understood in order for this project to be executed successfully.

2.3.1 Documentation

There will be a lot of study of documentation – for chosen tools and libraries, and for network protocols.

Due to the nature of the project, it is necessary to be clear about what various protocols do, how they do it, and what constitutes normal behaviour. For many protocols on the Internet, there are publicly available RFCs (request for comments) that are the canonical documentation for how they should be implemented[3].

All of the libraries and tools used in the implementation of this project have official documentation associated with them that explain how the tool or library is to be used. It will be essential to familiarise myself with this information.

2.3.2 Experimentation

Although the intended operation of a computer network is well documented, complex systems often exhibit unanticipated behaviour, and in any case it is the author's observation that one of the most effective ways to learn about a system is to interact and experiment with it. To this end, it is important to be open minded about incorporating ideas arrived at by experimentation.

[3] See the bibliography for several examples of RFCs and other standards referenced by this project.

Chapter 3

Implementation

In this chapter of the dissertation, I will explain how the system has been designed, and why it works in the way that it does. One of the continuing threads running throughout the design is the goal of modularity and its use to minimise the incidental complexity by allowing concepts to be implemented at a higher level and to manage the intrinsic complexity by breaking it down into smaller chunks. Many of the proceeding sections explain mechanisms that are used to attain this goal.

3.1 System Architecture

This section aims to explore the design of the project. It is clear that a project such as this can, and indeed should, be implemented in a modular fashion.

There is a wide variety of anomalies that one could wish to detect using this system. Therefore, I feel that modularity is key to this project. A plugin system that uses dynamic linking can be enabled at compile time to allow easy extension and run time configuration of used modules. Static linking of the plugins is permitted, however, in order to allow the system to run in a more limited environment – such as one with less storage, or one without support for shared libraries.

The use of a plugin system with dynamic valued events[1] allows for a high degree of decoupling[2]. Most importantly, in most cases the inter-dependencies between modules are described entirely by the events published

[1] Events are discussed in Section 3.3.1. Dynamic values are discussed in Section 3.3.3.

[2] There are a number of papers that have noted the very loose coupling produced by event systems, or publish/subscribe systems, such as the one I propose, some examples of which are cited[22][24][26].

and subscribed to.

This design also permits substitution of one module for another handling and publishing the same events. For example: during testing, the sniffer module may be replaced using a run time configuration option with a one that provides simulated network traffic from another source. I used this ability to write a test module to replace the sniffer with a mechanism that interacts with a suite of Python scripts. Through doing so, I was able to test the system with a simulated network that has behaviour I can specify exactly, and that provides much more useful debugging information.

3.1.1 Components

Broadly speaking, this project is composed of the following:

- A set of modules for transforming the received traffic into a processed form that is more easily understood by other modules, or for identifying anomalous behaviour. The processed form is typically an event or sequence of events.

- A subscription mechanism for linking the different modules to each other. This mechanism enables the modules to communicate with each other without imposing the rigidity of statically defining how and where information is used in the module that produced it. This system permits modules to receive information from several different modules to cater for scenarios where information only becomes apparent from analysis of information from several sources.

- A mechanism to allow the user to configure the system. This mechanism allows the user to specify information that is used to allow the system to more precisely define what anomalous operation is and is not (for example, by specifying that certain behaviours are not anomalous in the current environment, even though they would typically be so elsewhere). The user is also permitted to prevent the system from exhibiting certain behaviours – such as performing traceroutes – either by configuring the relevant module not to generate this traffic, or by actually specifying at run time that the module should not be loaded.

- A library of data structures and algorithms that may be common between many modules. Examples include:

  - Finite State Machines. These are a useful tool for implementing many different modules (for example, TCP).

– An information storage mechanism. This could be responsible for maintaining cached information which could either be compared later, or used to avoid unnecessary repetition of expensive operations. This mechanism might provide special facilities, for example to log old data and allow it to expire in the cache. This is implemented using SQLite and is described in Section 3.3.5.

3.1.2 Event Subscription Graph

It may be helpful to consider a first approximation design for how the system should process data received from the network. This is a particularly important aspect of the system, as it determines much of the overall architecture. To that end, Figure 3.1 shows part of a graph of possible events and module dependencies[3].

The graph is a directed graph with an edge from node A to node B representing an event that A could produce and B would receive. The diagram is coloured to make clearer the different kinds of module. The blue nodes are responsible for interacting with the outside world – for example writing to a log file, or listening to the network. The green nodes are modules for recognising received data. The yellow nodes generate traffic on network to gather information about it that could not be obtained by passive methods – in this example, by performing traceroutes. The red nodes detect anomalies[4].

Of course, there is no restriction on how many modules may subscribe to an event. Therefore, it is possible to easily add extra functionality, such as to log how many Ethernet frames are sent and received simply by allowing another module (in this example, a log) to receive these events. This graph may even contain cycles to permit the processing of encapsulated packets – for example, to permit IPv4-in-IPv4.

The subscription mechanism is in charge of maintaining the edges in this graph so that the system can be constructed in a much more modular way than would otherwise be achievable. Modules subscribe to events they would

[3] This graph has many omissions and a few extras compared to the actual system. It is intended to convey the ideas behind how the system could work, rather than as an absolute reference of exactly what modules have been implemented.

[4] Note: I have included "TCP/HTTP Mismatch" because it is a good example of how an anomaly may interact with more than one module. Suppose the system were to perform TCP fingerprinting (this would be implemented by a separate module, probably using an existing library) to identify the probable operating system of the machine from which the TCP stream originated. The system could compare this with any operating system claimed or implied by the HTTP's server header. If the two mismatch (such as Microsoft's IIS in a TCP stream from a Linux server), an anomaly event could be produced. I would need to do further research before I could implement this though.

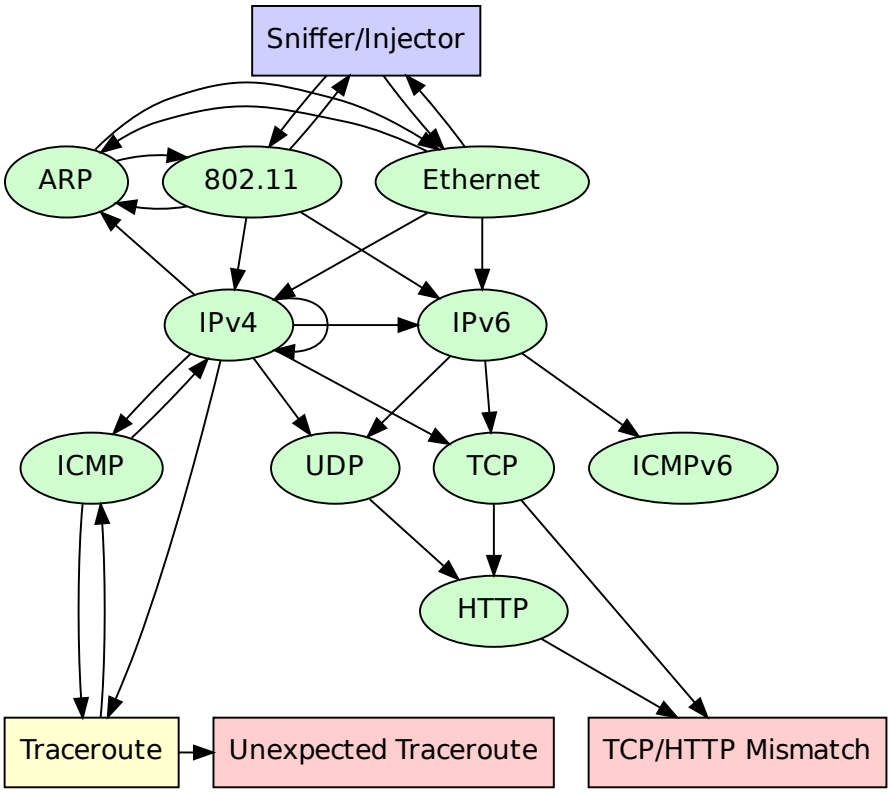

Figure 3.1: Partial Event Graph

like to receive using the subscription mechanism. This enables a new module to be added to the system without making changes to other parts of the system that would otherwise be necessary to link the new module in. The key here is to ensure this subscription mechanism is sufficiently flexible to accommodate an event model that provides tidy linking of loosely coupled modules. It becomes clear that a good implementation of this subscription mechanism is important.

This graph omits one important module – the logging module. The logging module subscribes to every event, and so any module that emits an event interacts with the logging module.

As an example, if a UDP in IPv4 in Ethernet packet is received, the sniffer would produce an Ethernet frame received event. Since the Ethernet module subscribes to Ethernet frame received events, this would be processed by the Ethernet module, which would then produce an IPv4 packet received event (as well as an Ethernet type 0x0800[5] packet received event). The IPv4 module (which subscribes to the IPv4 received event) would process this packet, and produce a UDP packet received event which will be processed by the UDP module.

To continue the example to demonstrate how the system is capable of sending data, suppose the IP address of the received packet is one that has not previously been encountered. The traceroute module would produce a send ICMP event. The send ICMP event contains a template send IPv4 event. When the ICMP module receives the send ICMP event, it uses the template send IPv4 event to produce a completed send IPv4 event with the payload data set to be an ICMP packet constructed from the send ICMP event's values. The IPv4 module receives the send IPv4 event, produces a layer 2 payload (much as the ICMP module produced an IPv4 payload), and sends an ARP send event. The ARP module receives the ARP send event, and looks up the Ethernet address for the IP address in the event. It produces an Ethernet send event with the appropriate Ethernet address. This approach is taken over sending layer 3 packets[6] in order to allow the system to remain independent of the operating system. This is particularly relevant if we use the system to detect anomalous behaviour of the local machine, however this is an extension to the project that is discussed in Chapter 5.

[5] Ethernet type 0x0800 tells the network stack that the protocol of the Ethernet packet's payload is IPv4[3].

[6] The standard sockets interface allows a program to open an IPv4 socket – specifying the protocol used, rather than a TCP or UDP socket.

3.2 External Code and Tools

This project has provided several opportunities to make use of external tools and libraries. This section documents which external libraries were chosen, and why.

PCRE[8] is used for the processing of regular expressions. Regular expressions are a powerful way to allow the user to configure which events should be logged and presented. They also represent a useful mechanism to implement recognition and processing of protocols such as HTTP. PCRE was chosen as it is a well used (and hence well tested) library, and uses regular expressions of a form that is widely understood. This should make it easier for experienced users of other tools to learn to use this system.

SQLite[16] is used to provide database support for modules that require it. It is small enough to use on embedded platforms such as a phone or a domestic gateway, and requires no complicated setup. The database support is used to allow persistent storage. This is useful, for example, to detect changes in routing over time.

Flex and Bison is used to produce the parser for the configuration file processor. This is explained further in Section 3.3.2.

Although written by me, the error encapsulation class is (with a few modifications) copied from other projects I have worked on in my spare time. I have chosen to use it because I have found it provides a useful mechanism to capture error messages from their point of origin, and display them – optionally with debugging information. This class is explored in Section 3.3.6.

3.3 Framework

A significant part of this project is the design and construction of a framework suitable for analysing packets, and determining the presence of anomalous behaviour of the network. This framework provides a mechanism to allow modular implementation of the project, a library of tools that are available to modules, and a mechanism to allow modules to communicate effectively with each other without the need for tight coupling. This section concerns itself with this framework.

3.3.1 Event System

Inter-module communication is achieved through dynamically typed events. Modules are able to subscribe to events based on their type and content.

They are able to do this at run time. This allows, for example, the HTTP module to subscribe to new data provided by a specific TCP connection.

By using dynamically typed events[7] rather than statically typed events, it is possible for the framework to provide a library of subscription factories that may simply be reused by the modules. Further, the dynamic typing permits modules to communicate with each other without the need for shared header files, or the need to maintain strict binary compatibility – if a module is updated to provide new information not used by another module, the second module need not be recompiled or altered in any way. The disadvantage of this is, of course, the lack of the thorough checking (of, for example, access to event members that exist) that C++'s type system would provide. It was also discovered during the early stages of design and implementation that trying to use C++'s type system for inter-module communication objects leads to a lot of very verbose code (and therefore an unnecessarily high incidental complexity).

The dynamic type allows for values that are key/value pairs. Each event is typically a set of key/value pairs, with each value being a further set of key/value pairs. This allows for compound events (with multiple types), and event types that may be shared between modules without the need to include a common header in both. Figure 3.2 demonstrates an example event. This event is a compound event – with a received:ethernet and a sniff event. The sniff event contains the keys: data, and interface, the values of which are not included in the diagram. Similarly, the received:ethernet event contains the key data[8].

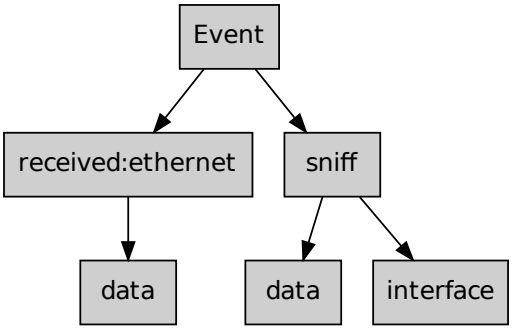

Figure 3.2: Example Event

Since events are defined by key/value pairs, it is possible for logging modules to filter events based on their name, or the values of various fields, based only on configuration information – without the need for the logging module to have hard coded knowledge of possible events.

There are a number of libraries that could be used to implement a modular design such as the one proposed in the Proposal and Section 3.1.2. In

[7] The dynamic type system used is described in Section 3.3.3).

[8] If it appears odd that this event does not contain any Ethernet addresses, note that this event is subscribed to by the Ethernet module. The Ethernet module produces a processed:ethernet event.

particular there are a number of message oriented systems. Most of these seem to be significantly more complex than required for my purposes, and are largely focused on implementing distributed systems – which is not the focus of this project. If it becomes desirable to transmit events over the network, they could be serialised and transmitted using a TCP socket by a module in the system.

Event handler objects subscribe to events, and nominate an event handler method to be called when a subscribed event is published. Subscriptions are handled by registering subscription objects with the event system. This provides a more structured alternative to simply performing the desired pattern matching in each event handler. This structured alternative permits greater code reuse – including the use of data structures such as finite state machines, and a library of useful subscription factories.

3.3.2 Configuration

The configuration mechanism allows a list of text configuration files to be specified on the command line, and processed in order. Configuration options in subsequent configuration files override or extend (for example, by concatenation) the options in previous ones. This enables information to be split between several files (such as to enable information about known bad HTTP headers to be in its own configuration file), and to enable common configuration with specific configuration options for several different networks where a machine is portable (or indeed to facilitate debugging).

In order to permit many different modules to use the configuration system (and indeed to enable the configuration to specify which modules are loaded), the configuration system uses run-time defined naming. The configuration format also supports many of the data types supported by the dynamic values (described in Section 3.3.3) – including lists and key/value pairs.

Unfortunately, I could not find a configuration library supporting all of these features[9], and so the writing of a custom configuration processor was required. To write the lexer and parser, I used Flex[4], and Bison[2]. A tree of dynamic values is created that may be accessed from any module – modules are able to simply read values from the configuration mechanism.

This has proven to be a versatile way to allow configuration information from several sources (such as a generic configuration for all networks the machine is typically connected to, and configuration files for specific networks) to be easily combined.

[9] libconfig[6] supports all except option overriding and extension.

Type	Description
Integer	A 64 bit integer.
Floating point	A floating point number of type 'double'.
String	A textual value.
Pointer	A pointer. This is included because a few modules are required to exchange pointers to structures that are not suitable for conversion to a dynamic value (such as during the process of registering a socket module with the framework).
Binary data	A 'buffer' object. This is described in Section 3.3.4
Event	In some cases, it may be that an event should be published at a future time. To accommodate this, the dynamic type supports the storage of an event.
List	A list of dynamic values.
Map	A set of key/value pairs where each key is of string type, and each value is of dynamic value type.

Table 3.1: Types that may be contained within a dynamic value

3.3.3 Dynamic Values

The dynamic value class provides a common representation of data throughout the system, and is vital to the operation of the event and configuration systems. It contains a lot of the incidental complexity that would be removed had a higher level language been chosen, however it has been implemented in a way that presents this complexity only once, thereby providing the rest of the system with the advantages of a higher level typing system, with few of the disadvantages. A dynamic value may have any one of the types described in Table 3.1.

It is possible to specify the formatting of the dynamic value when converted to a string. This is useful to allow logging modules to automatically produce output that is appropriately formatted without the need for prior knowledge of how values should be represented in text, or the need to define separate data types where only the formatting is different. An example of its use is the representation of IPv4 addresses – which are stored in a binary data object. The IPv4 module sets the format of the value. When events with an IP address are displayed by a logging module, the binary data is converted to the recognisable dotted quad notation.

The dynamic types are achieved through the use of a set of classes that implement a specific type, and a class that represents a dynamic type, and has a pointer to a specific type object. Although the dynamic type classes

are less tidy than might normally be desired, this is a consequence of placing a significant amount of complexity in the same place. The benefit is that this complexity is present nowhere else, and may simply be reused. The effect is much the same as having used a higher level language such as Python – but with the advantages of superior portability and performance. Exposed to the rest of the program is a single dynamic type class. This class has proven to be both suitable and convenient for communicating data within the system, and appears to be reliable.

3.3.4 Binary Data Buffer

This structure stores arbitrary binary data in a way that takes care of memory management. It is used in many places throughout the program to represent binary data such as payloads, machine addresses (usually IP addresses and MAC addresses), and so on.

The buffer class is responsible for allocating and deallocating memory on the heap. Since buffer objects are typically allocated on the stack, the incidental complexity of manual deallocation is removed, along with a significant chance of memory leaks. Allocation on the stack is possible even in cases where this would be difficult using traditional C++ stack memory – such as where concatenation of data of unknown lengths is required. There is little performance penalty associated with the use of the buffer class. In fact by using reference counting, it reduces the amount of copying that must be performed.

This structure also supports a variety of useful operations, such as concatenation, and endian translation[10].

In order to allow in place computation of the checksum used by the IPv4 module, the algorithm specified in RFC 1071[9] is implemented in the buffer class[11].

The framework also provides a buffer into which out of order data may be placed for reassembly. This is useful for modules like TCP that work with potentially out of order data. Although this is currently used only by the TCP module, it was felt that other modules might benefit from this facility in the future – the purpose of the library being to provide a set of tools that modules may opt to use, in addition to a set of common structures for

[10] Although this could be implemented as a stand-alone function, in practice this operation is performed exclusively on data stored in a buffer. It has therefore proven more convenient to allow the buffer to provide access to data with appropriate endian translation.

[11] In fact, the RFC provides a C implementation which I adapted for use in the buffer class.

communication.

3.3.5 Persistent Storage

A database class is included in the framework to allow modules to take advantage of persistent storage. The decision to include the database interface in the library rather than as a module (that would have permitted alternate database systems to be used more easily) has permitted sequential operations on the database to be performed without the need for a very large number of event subscriptions – an option that would have proven unduly inconvenient. The database class is essentially a C++ wrapper for SQLite that makes use of my dynamic type system. SQLite was chosen as it is a well tested library that is suited to storing diverse medium sized data – even on relatively small platforms. Using SQLite avoids the need to implement a completely bespoke (and probably buggy) storage mechanism.

3.3.6 Error Reporting

The framework provides a mechanism to produce error exceptions containing a formatted string in response to error conditions[12]. Occasionally an error may signal a normal condition that requires an alternate program flow[13].

The error reporting class provides a printf() like constructor that is extremely useful for providing information in error messages. Error exceptions may be caught in one of several key places in the program – such as in the event handling code. Processing of the packet may be halted, or the program terminated, depending on where the error is caught (and thus where it originally occurred).

Through the use of a macro that is usually used to call the constructor to the error class, the file and line of an error can be included in error messages in debugging builds of the software. This provides a versatile way to handle error conditions within the program.

3.3.7 Finite State Machine

FSMs (finite state machines) are a useful abstraction for the implementation of the event subscription mechanism, as well as several different modules (such as TCP and HTTP). The classes provide a way to divide complex code

[12] In fact I adapted a class I originally wrote for another project, but that I have found useful in several others for this purpose.

[13] This occurs where the type of a dynamic value must be determined by the SQLite interface.

into manageable chunks, while keeping track of some of the current state without the need for clumsy enums[14]. This is beneficial for good software engineering, and has proven to work well in practice. The class extends the notion of an FSM to provide a powerful mechanism for sharing data between states[15].

The FSM library is implemented as a set of abstract classes with methods that should be overridden to process "characters" (a "character" is an object of the same type as the FSM processes; often this is an event). FSM states provide a method to process a character, returning the next state. This method is passed data that may be shared between all of the states in the FSM.

When a character is processed, the FSM state may perform some complex action on the character. For example, the TCP module adds received data to a buffer in the shared data when in a state that permits TCP to transfer data.

3.3.8 Sockets

In order to allow several modules to be waiting for data to arrive over a socket (or file), the framework provides a mechanism for managing sockets. The module registers the file descriptor of the socket with the socket manager. When the socket is available for reading, the socket manager calls a virtual method of the socket module. The socket management is integrated with the timers as described in Section 3.3.9.

3.3.9 Time and Timers

There are a number of modules (such as the traceroute module) that depend on accurate timing. To accommodate this need, the framework provides a class to represent a time – including the current time. The time class is available as a dynamic type, as well as a stand alone class.

The class represents time as a UNIX timestamp[16], but with a granularity that may be specified at compile time[17].

[14] An 'enum' is a mechanism provided by C++ to allow the compiler to assign a unique integer to a name that may be used as a constant in the code.

[15] Although this means these are not pure FSMs, the ability of different states to access a common object is very valuable for many tasks that are neatly represented as an FSM with this extension.

[16] A UNIX timestamp is a representation of a time that is a signed 32 bit integer number of seconds since 1st January 1970, 00:00 UTC.

[17] This solution averts a "year 2038 bug" style problem, while permitting an increase in resolution should it be required – without hard-coding an indivisible unit of time.

Further, there are a number of modules that require delayed action. This is useful to detect, for example, timed out traceroutes. To achieve this, a timer system is implemented that causes the socket manager to wait only a finite period of time for a socket to become readable.

Each timer has a virtual handler method that is overridden. When a socket becomes readable, or the specified time elapses, the timer manager executes the handler of each expired timer. It then calculates the time until the next timer expires, and supplies this to the socket manager.

Although this is implemented as a module that listens to events just as any other, this is part of the library due to the unusual interaction with the IO loop.

3.4 Modules

This section describes the modules that have been implemented. These modules make use of the framework described in Section 3.3. Most of these modules produce events that can be subscribed to by other modules. This section also describes some of the interactions between the modules.

The project's modules may be compiled either statically into the main program, or as dynamically loaded modules. In the latter case, it is possible to specify at run time which modules are loaded – for example to swap the sniffer module for a testing module.

3.4.1 Address Resolution Protocol

The ARP (address resolution protocol) module serves two purposes. Firstly, like any other dissection module, it allows the system to interpret the protocol it implements – namely ARP[13]. Secondly, ARP is the protocol which translates IP addresses into the hardware addresses required to write traffic onto the network over a raw socket. This module is therefore required by the IPv4 module's send functionality. This functionality is in turn used by, for example, the traceroute module. When an IPv4 packet is to be sent by the system, the IPv4 module sends an event that is recognised by the ARP module. After determining the hardware address of the destination IP address, the ARP module generates an event for the Ethernet module. In addition to providing events required to detect anomalous ARP traffic, this module provides benefits that are explained in the last paragraph of Section 3.1.2 when sending IPv4 packets.

3.4.2 Ethernet

This module is responsible for dissecting Ethernet packets, and for adding Ethernet packets to IPv4 packets that are sent over the network. This is a relatively simple task that uses one of the structures provided by the program's library to translate the EtherType[18] into an Ethernet processed event.

It is worth noting that there is a standard for VLAN tagging [1] that is not implemented by this module. Such a module could be easily implemented to recognise the appropriate EtherType (0x8100[3]) and produce a second Ethernet received event without the VLAN tag.

3.4.3 Internet Control Message Protocol

The ICMP (internet control message protocol) module is responsible for encoding and decoding ICMP packets. It is required for the traceroute to work. It also produces anomaly events for unknown ICMP types and codes.

3.4.4 Internet Protocol Version 4

The IPv4 (Internet protocol, version 4) module is responsible for encoding and decoding IPv4 packets. Packets may be sent via the IPv4 module as described in Section 3.4.1. Like the Ethernet module, the IPv4 module uses the structure provided by the program to produce an IPv4 processed event. Unfortunately, there is a layer violation between IPv4 and some transport layers (such as TCP[12] and UDP[10]). These include a checksum that is based on the IPv4 header[9]. To facilitate this, the IPv4 module generates a piece of binary data that is used by these modules in the implementation of their checksum. It is expected that IPv6 could be implemented in much the same way as IPv4.

3.4.5 Logging

In Section 3.1.1, we discussed the need for a logging mechanism. As all information that we may wish to log is expressed in the form of events, the logging mechanism may be implemented as a module like any other. It would be possible to have multiple loggers presenting different information to different destinations (such as to a GUI, terminal, network service, and so on), however only a single configurable logger that outputs to standard out has been implemented.

[18] Ethernet frames have a number associated with them, known as the EtherType. This number specifies the type of packet that is in the payload.

The logger can be configured to filter the output according to the event type name, or contents of the event. The filtering is done by regular expression implemented with PCRE[8]. Using this filtering, the user may specify that particular anomaly events are in fact expected, or that understood events are in fact anomalies.

In order to permit the user to identify the results of each packet, the logger splits the stream of events by packet sniff events. The user may configure events to be excluded from the split result.

3.4.6 Port Scan

Section 2.1.2 defines a port scan. A port scan can be detected by counting the number of unsolicited accesses by a given IP address to any TCP or UDP port. If this number becomes so high that it is likely to be the result of a deliberate attempt at performing a port scan, the module produces a port scan anomaly event.

The port scan module subscribes to TCP packet processed events and UDP packet processed events. Each IP address that either sends or receives TCP or UDP packets to or from the local machine has associated with it a record containing the set of ports that the local machine has used as a source port with that address, and the set of ports that the remote machine accessed that were not in the set of source ports. If the size of the second set grows to beyond a specified number, the module produces a port scan anomaly event.

3.4.7 Resource Monitor

The resource monitor module subscribes to interface send events, and interface sniff events. It is used in order to collect much of the data used to evaluate the project in Chapter 4. The resource monitor module records to a file information about the CPU, memory, and network usage of the system each time a packet is sent or received.

3.4.8 Sniffer

It is the sniffer module that acquires data from the network, and writes packets generated by the program onto the network. This is implemented as a module to allow it to be replaced with a testing module that provides a simulated network. It could also be replaced with a module that permits the program to analyse the network traffic of another machine.

3.4.9 Named Pipe Testing

In order to allow the program to be tested using simulated network traffic, I have implemented a module that replaces the sniffer module. The replacement module reads packets from a named pipe[19], and writes packets to a different named pipe. I have used this to interface with a set of Python scripts, however I could use any language that can read from and write to files. This is a good example of the interchangeability advantages of a modular approach.

The interactivity afforded by this named pipe approach permits the test suite to generate packets in response to those generated by the system – something which would not be possible using a packet dump that had been prepared beforehand.

3.4.10 Traceroute

This module implements an ICMP traceroute. It stores the results to disk to allow the module to forgo repeating traceroutes that were performed a previous time the program ran. When a traceroute is complete, it produces an event with details of the address tracerouted, and the nodes that appear as intermediate routers. A traceroute operates by sending packets with increasing TTL (time to live). Since a router is required to decrement the TTL of a packet it routes and is required to emit an ICMP Time Exceeded packet if the TTL becomes zero[11], it is possible to perform a traceroute by producing packets[20] with increasing TTL. As a result of difficulties that are described in Section 4.3.1, an option was added to prevent the module from tracerouting IP addresses that are contacted by ICMP only.

A second module is implemented that detects anomalies in the traceroute. It simply listens for traceroute events, and determines whether they contain intermediate nodes that are known to be bad (such as those known likely to be censorship proxies[25]), or have a node with an unusually long hop time[21]. It would be easy to extend the system by implementing more modules that detect other possible anomalies in the traceroute.

[19] A named pipe is a file that acts like a FIFO – written to by one process, and read from by another.

[20] It is possible to use any IPv4 packet with the TTL properly set, however I have chosen to use ICMP Echo Request packets.

[21] This is the time between the sending of the ICMP Echo Request packet and the receipt of the ICMP Time Exceeded (or ICMP Echo Reply) packet. This is chosen over comparing the difference from the previous hop for simplicity, and because it is equally as effective at detecting the intended anomaly.

3.4.11 Transmission Control Protocol

This module is responsible for decoding TCP (transmission control protocol) segments, and for producing events that specify the reception of a TCP segment. The module is able to produce information about the segment, such as the source and destination ports, and the flags that were set.

3.4.12 User Datagram Protocol

This module simply removes the UDP (user datagram protocol) header, and generates an event with the appropriate source and destination address and ports, and the payload.

3.5 Summary

The system is implemented in a very modular way, using a publish/subscribe event system with dynamically typed events. The result is a very powerful framework, well suited to the task of detecting anomalous behaviour, upon which a suite of modules have been developed.

Chapter 4

Evaluation

4.1 Detection of Anomalies

The first criterion from the project proposal[1] discusses the requirement that the project be able to detect anomalous behaviour of the network. It also suggests some anomalies that the system should be able to detect to demonstrate the correct operation of the system.

4.1.1 Unusual Traceroute Path

The system is able to perform traceroutes on IP addresses that are contacted, and identify nodes that appear in a blacklist. Appendix A shows the output that results when using a web browser to visit a website (the homepage only, on my Virgin Media connection) that appears in the IWF's list of sites to block. This part of the criterion is fulfilled.

4.1.2 Hop Time

Using the traceroutes collected as described in Section 4.1.1, the system can identify nodes that responded to the traceroute packets in an unusually long

[1] "The system should be able to detect a number of different anomalies, using active and passive techniques where appropriate. These anomalies might include:

- Unusual path revealed by traceroute
- Unusual time taken to reach a hop
- Port scanning of the local machine
- The receipt of unusual packet types"

time. An example of the resulting output is presented in Appendix B. The threshold round trip time was set to 100 ms to make testing easier, however this is simply an option in the configuration. This part of the criterion is fulfilled.

4.1.3 Port Scanning

The system can process TCP and UDP packets to produce events that identify the source and destination addresses and ports. The port scan detection module is able to process these events to identify machines that attempt to access too many ports without solicitation. Appendix C demonstrates the output produced by the system when a port scan is detected. The port scan is the default SYN scan performed by Nmap[7]. Port scans are successfully detected; this part of the criterion has been achieved.

4.1.4 Unusual Packets

When the system encounters a packet that is not known an anomaly event is produced. It is possible to filter these anomalies using the configuration for the logger, or to flag up packets that were understood. This is a good example of how the logging module can be configured based on the user's specification of what kinds of network traffic and behaviours are expected. Appendix D shows the response of the system to an unrecognised packet[2]. The system has discovered many packets that were previously unknown to me on several networks. This part of the criterion has been achieved.

4.2 CPU and Memory Requirements

This section addresses the second success criterion[3] suggested in the proposal. It is clearly important that the system be able to operate correctly and with useful results given reasonable finite resources. This section provides an analysis of the resources that are required by the system.

[2] This 6 byte Ethernet frame is produced by my Android phone. The same frame (albeit with a different source address) is produced by my father's Android phone. I believe it to be a bug, as the packet serves no apparent purpose, and conveys no apparent data (since it is always the same). If it is not a bug, then it is an anomaly that indicates an function of the phones that I did not know about – which is also something I want the project to detect.

[3] "An analysis of the memory and CPU requirements of the system, and how this is affected by the amount and type of network traffic received. It is desirable that the system can cope with a typical, but high load."

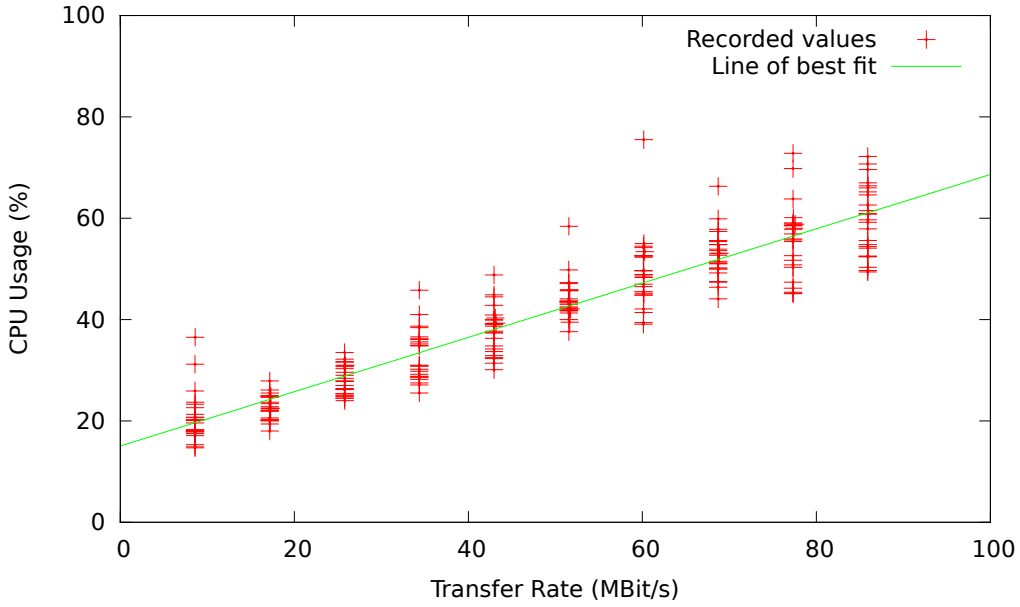

Figure 4.1: CPU Usage Under Load

Figure 4.1 shows the relationship between the transfer rate of incoming test data over a TCP stream, and the associated CPU usage of the local machine running the system. The program performed the experiment 25 times for each speed. The machine has a dual core processor, however the system currently uses only a single thread. This means the program itself can only use up to 50% of the CPU, although other consequent use – such as by the kernel, or other processes, may still allow the total processor usage to exceed 50%. This should be remembered when interpreting this graph. It is apparently possible for the system to handle a throughput of 50 MBit/s – albeit with significant (about 40%) load on the CPU. Higher rates are just about achievable, up to around 100 MBit/s. It should be noted that there is some noise in this graph – as it measures the CPU usage of the entire machine, not just the one program. When this test run without the program running, the CPU usage has a similar amount of noise, and runs at roughly 15% or 20%.

Figure 4.2 shows the memory usage as a function of time. The data for this graph was produced with a typical set of modules loaded, plus the resource monitor module. The system was allowed to run for several hours during which I engaged in normal activities with the Internet. Although the graph has the appearance of a memory leak, the memory is used by the traceroute and port scan modules, which accumulate data as the system

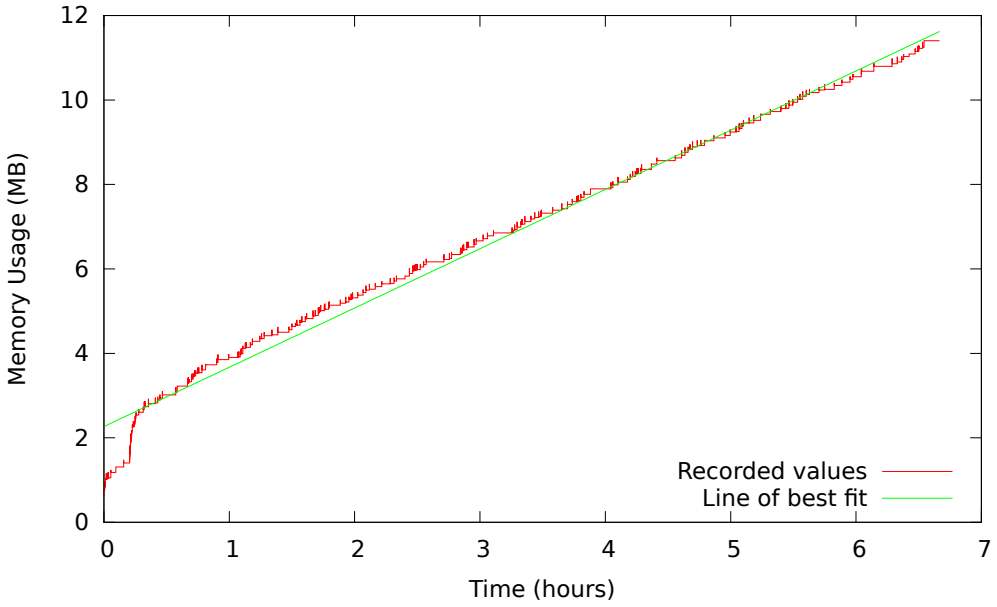

Figure 4.2: Memory Usage

runs. This data is necessary in order to allow the system to use data collected about past traffic to detect anomalies, or to perform a traceroute only if one has not been performed on that address. Given more development time, it should be possible to make these modules take full advantage of the persistent storage[4]. If this were to be done, it should be possible to keep the memory usage below a constant threshold. By extrapolation, it should be possible to run the system for over two days before it consumes 100 MB of memory. Although this would have to be fixed before the system will meet the strictest version of success criteria set out in my proposal[5], the system as implemented meets the basic version.

[4] Currently, the traceroute module keeps a set of traceroutes already performed in persistent storage in order to avoid doing traceroutes that have been done a previous time the project was run, however more work could be done to make better use of the storage in order to cache only some of this data in main memory.

[5] "Success for this category of criteria will require that the system should work well on modest end user hardware under typical load. Ideally, however, we would like that the system can provide most of its functionality in much more challenging environments. In this case, we may consider the project to be especially successful by these criteria."

4.3 Traffic Generated

4.3.1 Traceroute

The tracerouting performed by the program is particularly worthy of commentary in this evaluation, as several issues arise to complicate successful collection and use of this data. The traceroute module, at its simplest, is designed to perform a traceroute on every IP address with which the local machine has contact.

The approach initially attempted simply tracerouted every IP address from which a packet is received, or to which a packet is sent. This includes packets from intermediate nodes generated by tracerouting. This works if the network is close to (even if not exactly equivalent to) being a tree from the end system's point of view.

Unfortunately, when this method was attempted on a real network, there were sufficiently many routing anomalies that increased the number of intermediate nodes[6], that the queue of nodes to be traceroute increased much faster than they could be processed. Figure 4.3 gives a small example of a routing anomaly that produces this problem. In this example, if 203.0.113.168 or 203.0.113.169 is tracerouted, then the path appears to be via 198.51.100.5 and 203.0.113.4. If, however, 203.0.113.4 is tracerouted, the path appears to be via 198.51.100.6. With many of these anomalies, such as if the apparent path to 198.51.100.6 is also different, then the number of routers we attempt to traceroute grows very quickly. Even if we could perform all of these traceroutes, it is likely that no useful information could be easily gathered, as there would be a huge number of anomalies.

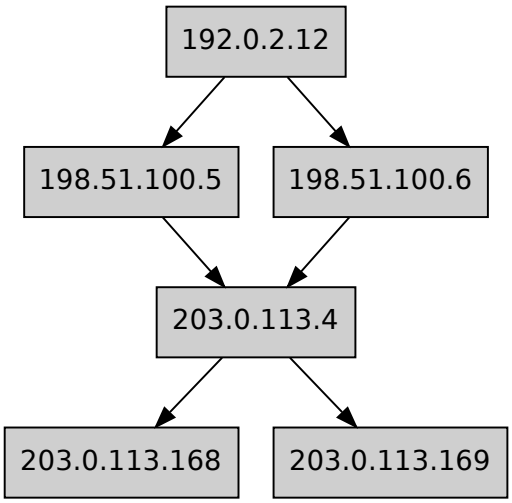

Figure 4.3: Routing Anomaly

If the user configures the module so that IP addresses are not tracerouted

[6] Although the idealised version of the Internet is a tree from the perspective of a single node, in practice, ISPs often use load balancing, and other non-obvious routing policies that mean the view may not be a tree.

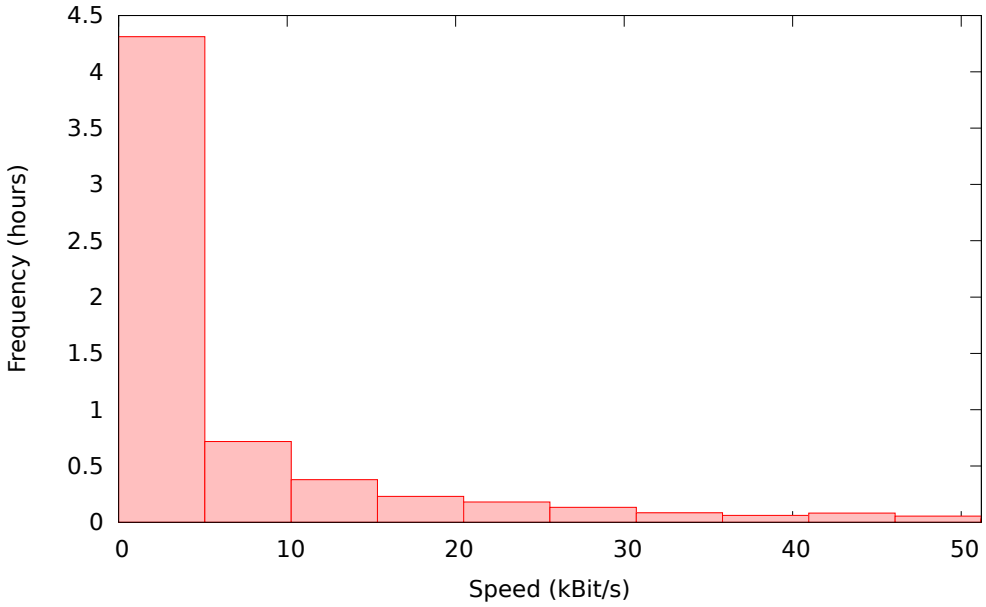

Figure 4.4: Histogram of Rate of Traffic Sent

if they are contacted in the form of an ICMP packet, the system behaves in a sensible way.

In the proposal a requirement was stated that the traffic generated by the system not be excessive[7]. Figure 4.4 shows the rate of sent traffic as a histogram. The graph demonstrates that this is usually less than 10 kBit/s[8]. In fact, the system sent less than 1 MB of traffic during the 7 hours the test was run. This is a very acceptable rate of traffic, and achieves the goal that the traffic should not be excessive.

4.3.2 Trade Offs

With such a diverse range of environments on which we may wish to run the system – from a small domestic gateway connected to a low bandwidth

[7] "It is desirable that the system should not generate unacceptable traffic under any circumstance." A significant component of "unacceptable traffic" is too much traffic. Since the project only generates ICMP Echo Request packets, which we shall assume to be acceptable if the user has enabled the traceroute module, the best analysis of whether this criterion is met lies in an analysis of how much traffic is generated.

[8] This speed is computed as an exponentially weighted moving average of the speed, with a half life of one second. The amount of time spent at higher than 50 kBit/s (the rate of traffic sent does not exceed 0.25 MBit/s) is extremely small, and has not been included in Figure 4.4.

network to a powerful server with a 10 Gigabit connection – it is important to consider the trade offs involved when choosing what analyses to perform. The traceroute module provides an excellent example of a module where there are many such trade offs.

Firstly is the trade off required to solve the problems described in the Section 4.3.1. It is possible to traceroute only a subset of the nodes contacted as a consequence of tracerouting. This prevents the queue of traceroutes from becoming ever longer, however less information about the network is collected.

In general, it is often possible to reduce resource or traffic requirements at the expense of reducing the accuracy or amount of information collected.

Another important consideration for the traceroute module (and for other modules that might, for example, require a cache of HTTP headers returned by HTTP servers) is persistent storage and its rate of access. Much of the information gathered could be stored in persistent storage, thereby conserving memory. In many cases, data structures such as a hashmap, could be used to allow common operations to be performed with a small amount of data in memory, while occasionally being forced to go to disk. This would be an extension.

4.4 Modularity and Extendability

It is a success criterion that the system should be implemented in a way that is modular and easy to extend. The framework represents the set of mechanisms that are used to achieve this last success criterion[9] listed in my project proposal. In this section, I will discuss how well decoupled the different modules are from each other, and how difficult it would be to implement a new module.

The trade off to use dynamic typed events as discussed in Section 3.3.1 has proven to be well worth it! The result is loose coupling between modules, and a tidy event system. This loose coupling is exactly what is required to enable new functionality to be easily added to the system – as required by the last success criterion.

4.4.1 Ease of Extension

In this section, I consider the amount of work that would be required in order to implement a new module for the system. This serves as a good

[9] "The system should be implemented in a modular way that would enable further functionality to be added easily."

example of managing incidental complexity (a topic discussed in the Software Engineering course) – leaving only the intrinsic complexity behind.

One potential source of incidental complexity might be changes that are required to be made to the rest of the system in order to interface the new module with the existing ones. Since the modules communicate with each other through dynamically typed events, it would not be necessary to change any other part of the system to accommodate the new module – even if the new module were to replace an existing one.

Since the framework provides a library of useful tools and data structures, it is likely that these can be used in the development of the new module – thereby reducing the amount of code that must be written. This avoids the need for ugly and difficult to understand hacks that result from naive reimplementation of functionality. It is also not necessary to test these structures upon use, as they will likely have been tested by other modules' use of them. Many of the structures (such as the binary blob class) provide a way for different modules to exchange information with one another.

The event system provided by the library is versatile, and allows modules to express information they're interested in with a common format. A module that detects anomalous behaviour can simply subscribe to the information it requires, process the information in an event handler, and publish an event if the anomaly is detected. Likewise, a new module can be implemented by subscribing to the received events that would be expected to contain the protocol's data, processing those events, and producing processed events. The complexity of dissecting the lower protocols is handled elsewhere, as is the complexity of delivering that information to the module.

As an example of the effectiveness of this modularity, consider the time taken to implement the first of the specified anomalies implemented. Much of the framework had to be implemented for all four of the anomalies – as well as several protocols. This work, and the first anomaly took several weeks to complete. Having completed this work, I was able to implement the other two anomalies in a single night, and test and fix them given only a few more! The source code for, for example, the detection of an anomalous traceroute, is extremely simple because it need only identify such a path from a traceroute that is already provided by a module elsewhere in the system. Adding a second anomaly, detection of an unusually high time taken to reach a hop, also takes very little code.

In short, the new modules are able to take advantage of functionality implemented by other modules, and by the framework, with as little incidental complexity as possible. It is easy to add new functionality to the system, and this was the aim of the final success criterion. This success criterion has been achieved well.

4.5 Summary

My project works well, and has achieved all the success criteria set out in the project proposal. It detects several anomalies, including all the ones specified as success criteria, and a number of other anomalies too. Of course, this project, being somewhat open ended, leaves plenty of scope for extension[10]. Section 5.1 discusses this further.

[10] As remarked elsewhere, it would be impossible to detect every conceivable anomaly in a Part II Project.

Chapter 5

Conclusions

This project has been completed successfully. All of the success criteria set out in the proposal have been met, with many features implemented. As a network anomaly discovery tool, it is somewhat pleasing to have discovered many unknown properties in the networks my machines are connected to – including genuine bugs, such as the 6 byte Ethernet frame emitted by my phone! To bring the project full circle, the project is capable of detecting the interception discussed in Chapter 1.

If planning the project with hindsight, I would probably converge much more quickly on a design that resembles the one chosen – it appears to be an effective solution. There are a number of small changes[1] I might wish to make, however it would be perfectly feasible to make these changes to the project as it stands.

5.1 Future Work

Of course, since much of the project is the production of infrastructure suited to real time analysis of received network traffic, this is not as far as the project could be taken. As a modular project with such an open ended basic aim, there is enormous scope to implement new functionality – extra anomalies, more protocols, and so on.

One could also seek to complete the functionality that I did not have time for. The project could easily be extended to process IPv6. This would be a matter of implementing the IPv6 protocol as was done for IPv4. Due to time constraints, the TCP module has less functionality than I had hoped. It

[1] These include using different namespaces (a C++ mechanism for separating the code out into different named sections) for the modules, and improving the library of FSMs made available for creating subscriptions.

would be interesting to implement TCP stream decoding (as opposed to just the segments as described in Section 3.4.11), and see what kinds of anomalies can be discovered by analysing higher layer protocols such as HTTP. I had hoped, as an extension, to perform a statistical analysis on the traceroutes to automatically identify unusual traceroutes. Nonetheless, unusual traceroutes can be detected if the node that makes it unusual is known. More use of the persistent storage mechanism could be implemented in order to improve the memory usage of the system, as described in Section 4.2. There is scope to make the system multithreaded – which would enable more processor intensive analysis on systems with processors that have a large number of cores. A more novel idea might be to use machine learning techniques to discover anomalous behaviour that would be difficult specify using a fixed rule set[17].

On top of this, the project could be expanded to discover other kinds of information that can be revealed through inspection of network traffic, including much more complicated analysis of traffic – something which my system is well suited to. This might include evidence of a compromised local machine (by detecting emitted traffic that the administrator has not intended), or for more research oriented projects such as producing a map of the Internet. Another possible avenue for expansion might be to permit several instances of this project to communicate[2] – forming a federation of machines that can identify anomalies that arise out of observing that a small number of machines receive a different network response to the rest.

5.2 Final Words

This project has achieved its original aims, and would make a good platform upon which further development could take place. It has revealed the kinds of interesting information that it was intended to highlight, and has helped me to learn more about the chosen subject. I hope to continue working on this project – that has already been such a success – in the future.

[2] The idea of using a distributed set of nodes to gather information about a network has been explored and found to be viable[27].

Bibliography

[1] 802.1Q – IEEE Standard for Local and metropolitan area networks – Virtual Bridged Local Area Networks. http://standards.ieee.org/getieee802/download/802.1Q-2005.pdf.

[2] Bison. http://www.gnu.org/software/bison/.

[3] EtherTypes. http://standards.ieee.org/develop/regauth/ethertype/eth.txt.

[4] Flex. http://flex.sourceforge.net/.

[5] IP Protocol Numbers. http://www.iana.org/assignments/protocol-numbers/protocol-numbers.txt.

[6] libconfig. http://www.hyperrealm.com/libconfig/.

[7] Nmap. http://nmap.org/.

[8] Perl Compatible Regular Expressions. http://www.pcre.org/.

[9] RFC 1071 – Computing the Internet Checksum. http://tools.ietf.org/html/rfc1071.

[10] RFC 768 – User Datagram Protocol. http://tools.ietf.org/html/rfc768.

[11] RFC 791 – Internet Protocol. http://tools.ietf.org/html/rfc791.

[12] RFC 793 – Transmission Control Protocol. http://tools.ietf.org/html/rfc793.

[13] RFC 826 – An Ethernet Address Resolution Protocol. http://tools.ietf.org/html/rfc826.

[14] Service Name and Transport Protocol Port Number Registry. http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt.

[15] Snort. http://www.snort.org/.

[16] SQLite. http://www.sqlite.org/.

[17] Matthew V. Mahoney; Philip K. Chan. Learning Rules for Anomaly Detection of Hostile Network Traffic. http://www.cs.fit.edu/Projects/tech_reports/cs-2003-16.pdf.

[18] Richard Clayton. Failures in a Hybrid Content Blocking System. www.cl.cam.ac.uk/~rnc1/cleanfeed.pdf.

[19] Richard Clayton. Technical aspects of the censoring of Wikipedia. http://www.lightbluetouchpaper.org/2008/12/11/technical-aspects-of-the-censoring-of-wikipedia/.

[20] Richard Clayton. The IWF Blocking List Recent UK Experiences. www.cl.cam.ac.uk/~rnc1/talks/090630-inex.pdf.

[21] Lilian Edwards. Content Filtering and the New Censorship. *Fourth International Conference on Digital Society*, pages 317–322, February 2010.

[22] Patrick Th. Eugster; Pascal A. Felber; Rachid Guerraoui; Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2), June 2003.

[23] Jayant Gadge; Anish Anand Patil. Port Scan Detection. *16th IEEE International Conference on Networks*, pages 1–6, December 2008.

[24] Pietzuch; Peter R. Hermes: A scalable event-based middleware. Technical Report UCAM-CL-TR-590, University of Cambridge, Computer Laboratory, June 2004.

[25] Wikipedia. Administrators' noticeboard/2008 IWF action. http://en.wikipedia.org/w/index.php?title=Wikipedia:Administrators%27_noticeboard/2008_IWF_action&oldid=451983662.

[26] Antonio Carzaniga; David S. Rosenblum; Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), August 2001.

[27] Sridhar Srinivasan; Ellen W. Zegura. Network Measurement as a Cooperative Enterprise. *Proceeding IPTPS 2001 Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 166–177, 2001.

Appendices

A Anomalous Traceroute Output

Visited URL: http://img201.imagevenue.com/[3] [4]

```
anomaly:traceroute:blacklist:
        destination: 72.55.191.38
        node: 195.182.178.150
        rtt: 0.020000
        source: 192.168.1.16
        traceroute: [
                address: 192.168.1.1
                rtt: 0.090000
                ttl: 1,
                address: 82.29.40.1
                rtt: 0.028000
                ttl: 2,
                address: 213.106.254.117
                rtt: 0.018000
                ttl: 3,
                address: 213.105.159.205
                rtt: 0.023000
                ttl: 4,
                address: 213.105.64.22
                rtt: 0.020000
                ttl: 5,
                address: 195.182.178.150
                rtt: 0.020000
                ttl: 6,
```

[3] This URL was suggested by Dr Richard Clayton who has a list of domains and IP addresses that are on the IWF list.

[4] This output has been trimmed to include only the relevant part.

```
        address: 62.30.0.204
        rtt: 0.019000
        ttl: 7,
        address: 62.30.249.46
        rtt: 0.022000
        ttl: 8,
        address: 213.161.65.149
        rtt: 0.031000
        ttl: 9,
        address: 64.125.14.18
        rtt: 0.021000
        ttl: 10,
        address: 4.69.139.120
        rtt: 0.033000
        ttl: 11,
        address: 4.69.153.129
        rtt: 0.024000
        ttl: 12,
        address: 4.69.137.78
        rtt: 0.089000
        ttl: 13,
        address: 4.69.134.66
        rtt: 0.089000
        ttl: 14,
        address: 4.69.141.5
        rtt: 0.096000
        ttl: 16]
ttl: 6
```

B High RTT Output

Produced by setting the RTT threshold to 100 ms[5].

```
anomaly:traceroute:rtt:
        destination: 60.241.203.71
        node: 38.104.138.94
        rtt: 0.389000
        source: 192.168.1.16
        traceroute: [
                address: 192.168.1.1
                rtt: 0.014000
                ttl: 0,
                address: 192.168.1.1
                rtt: 0.007000
                ttl: 1,
                address: 82.29.40.1
                rtt: 0.021000
                ttl: 2,
                address: 213.106.254.109
                rtt: 0.013000
                ttl: 3,
                address: 213.105.159.205
                rtt: 0.024000
                ttl: 4,
                address: 62.253.185.118
                rtt: 0.023000
                ttl: 5,
                address: 62.253.174.18
                rtt: 0.079000
                ttl: 6,
                address: 130.117.14.141
                rtt: 0.033000
                ttl: 7,
                address: 130.117.50.138
                rtt: 0.036000
                ttl: 8,
                address: 130.117.0.97
                rtt: 0.017000
```

[5] This output has been trimmed to include only the relevant part.

```
        ttl: 9,
        address: 154.54.43.193
        rtt: 0.115000
        ttl: 12,
        address: 154.54.82.141
        rtt: 0.125000
        ttl: 13,
        address: 154.54.24.109
        rtt: 0.163000
        ttl: 14,
        address: 154.54.1.130
        rtt: 0.165000
        ttl: 15,
        address: 154.54.6.238
        rtt: 0.165000
        ttl: 16,
        address: 38.104.138.94
        rtt: 0.389000
        ttl: 17]
ttl: 17
```

C Port Scan Output

Command: **nmap -sS 192.168.1.16**[6]

```
anomaly:portscan:too many unsolicited:
        count: 11
        destination: 192.168.1.16
        source: 192.168.1.55
```

[6] This output has been trimmed to include only the relevant part.

D Unusual Packet Output

Unusual packets apparently originating from my Android phone.

```
[*] logger:stack:
        logger:stack: [
                processed:ethernet:
                        destination: FF:FF:FF:FF:FF:FF
                        ethertype: 0x6
                        payload: Length: 46
                                00 01 AF 81 01 02 00 00 00 00 00 00 00 00 00 00
                                00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                00 00 00 00 00 00 00 00 00 00 00 00 00 00
                        source: 90:21:55:E6:F2:28,
                anomaly:table:ethernet:unknown:
                        buffer: Length: 46
                                00 01 AF 81 01 02 00 00 00 00 00 00 00 00 00 00
                                00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                00 00 00 00 00 00 00 00 00 00 00 00 00 00
                        key: 6]
```

E The Pirate Bay Interception

Visiting The Pirate Bay (http://thepiratebay.se/) on a Virgin Media Internet connection after they blocked it[7].

```
anomaly:traceroute:blacklist:
        destination: 194.71.107.15
        node: 195.182.178.150
        rtt: 0.068000
        source: 192.168.128.3
        traceroute: [
                address: 192.168.128.1
                rtt: 0.065000
                ttl: 0,
                address: 192.168.128.1
                rtt: 0.038000
                ttl: 1,
                address: 192.168.1.1
                rtt: 0.040000
                ttl: 2,
                address: 82.29.40.1
                rtt: 0.084000
                ttl: 3,
                address: 213.106.254.109
                rtt: 0.099000
                ttl: 4,
                address: 213.105.159.205
                rtt: 0.068000
                ttl: 5,
                address: 213.105.64.22
                rtt: 0.077000
                ttl: 6,
                address: 195.182.178.150
                rtt: 0.068000
                ttl: 7,
                address: 62.30.0.204
                rtt: 0.077000
                ttl: 8,
                address: 62.30.249.46
```

[7] This output has been trimmed to include only the relevant part.

```
        rtt: 0.066000
        ttl: 9,
        address: 213.161.65.149
        rtt: 0.086000
        ttl: 10,
        address: 213.248.76.85
        rtt: 0.094000
        ttl: 11,
        address: 80.91.247.91
        rtt: 0.107000
        ttl: 12,
        address: 80.91.250.148
        rtt: 0.114000
        ttl: 13,
        address: 80.91.249.204
        rtt: 0.167000
        ttl: 14,
        address: 80.91.253.237
        rtt: 0.111000
        ttl: 15,
        address: 80.239.128.170
        rtt: 0.123000
        ttl: 16,
        address: 82.96.1.161
        rtt: 0.129000
        ttl: 17,
        address: 192.121.80.155
        rtt: 0.103000
        ttl: 18,
        address: 192.121.80.181
        rtt: 0.152000
        ttl: 19,
        address: 194.14.56.2
        rtt: 0.121000
        ttl: 20]
ttl: 7
```

Nicholas Tomlinson
Robinson College
nst25

Computer Science Tripos Part II

Network Anomaly Discovery

20th October 2011

Project Originator: Nicholas Tomlinson

Resources Required: See attached Project Resource Form

Project Supervisor: David Evans

Signature:

Director of Studies: Anuj Dawar

Signature:

Overseers: Cecilia Mascolo and Larry Paulson

Signatures:

Introduction and Description of the Work

My project will discover anomalous behaviour of computer networks by analysis of traffic reaching an end machine, and by sensibly selected generation of traffic from this machine (for example, traceroutes). This would be done by constructing a framework with which specific anomalies could be recognised, logged, and displayed to the user. It should be noted that, since the project is intended to be run on an end machine rather than elsewhere in the network, anomalies centre largely around the machine in question. For example, the detection of port scanning would typically be a port scan against the particular machine running the program.

Resources Required

- My laptop for development and testing. This is required as I must have root access on machines for testing. In the unlikely event this machine becomes unavailable, there are alternatives available that are otherwise non-essential.
- Internet access on the above for testing and research (this Internet access should not be unreasonably sensitive to unusual (but harmless) traffic - for example, a traceroute of every IP address that I would otherwise contact normally).
- Free open source software packages and libraries. These software packages and libraries may provide a number of useful facilities that my project will depend upon.
- There are several other resources that may be useful for the completion of this project, but on which my project's successful completion does not depend. These may be requested on an "if possible" basis, and are not listed here.

Starting Point

I have previously experimented with a somewhat primitive (both in design and in provided functionality) system in Python to detect a few specific oddities (primarily evidence of unexpected HTTP proxies, and mismatch of IP address and MAC address). This system has only fairly minimal modularity and functionality. I have written a very simple packet sniffer to become

familiar with the API for doing this (the program does nothing more than print a list of received packets with a few details about them). I have also experimented with HTTP header fingerprinting using machine learning (which was partially successful), and have a general interest in networking and have explored a few other projects that I feel are insufficiently related to this one to mention further.

Substance and Structure of the Project

Components

There are a wide variety of anomalies that one could wish to detect using this system. Therefore, I feel that modularity is key to this project. It is well worth noting that in this context, an anomaly is any traffic or behaviour of the network that is not expected. Typically, this will be unusual or non-compliant behaviour or traffic, however user specification may also dictate that traffic that would otherwise be considered normal is in fact anomalous, or that traffic that would normally be considered anomalous is in fact normal. An anomaly may obviously be be malicious (such as a port scan, or traffic interception), however it may also be unintentional behaviour (such as the Ethernet frame with a six byte payload that is occasionally produced by my phone, I believe erroneously) or just a curiosity (such as a change in network configuration).

Broadly speaking, this project might be composed of the following:

- Protocols. These protocols are responsible for interpreting the data received over the network.
- Anomalies. These anomalies are responsible for analysing the information provided by the protocols, and determining whether this information is indicative of anomalous behaviour. It is worth noting that anomalies, and indeed several other parts of the system are not dissimilar to protocols – both may interact using the subscription mechanism (explained in more detail below).
- A subscription mechanism for linking the different protocols to each other, and for linking anomalies to protocols. This mechanism should enable the different protocols used in a packet (or indeed, at higher levels, a stream) to be connected without imposing the rigidity of statically defining each link. This system should also permit anomalies to receive information from several different protocols to cater for anomalies which are visible only when examining multiple protocol layers in

a packet or stream. The subscription mechanism would enable different protocols to be notified of events generated by other protocols, anomalies, and other parts of the system.

- A mechanism to allow the user to configure the system. This configuration might allow the user to specify that the system should not perform certain operations (for example, the covert user might wish the system not to emit any additional traffic). This mechanism could also be used to allow the user to specify information that would allow the system to more precisely define what anomalous operation is and is not (for example, by specifying that certain behaviours are not anomalous in the current environment, even though it would typically be so elsewhere).

- A library of data structures and algorithms that may be common between many anomalies and protocols. Examples might include:

  - Finite State Machines. These are a useful tool for implementing many different protocols (for example, TCP) and anomalies.
  - An information storage mechanism. This could be responsible for maintaining cached information which could either be compared later, or used to avoid unnecessary repetition of expensive operations. This mechanism might provide special facilities, for example to log old data and allow it to expire in the cache.
  - A set of algorithms for performing statistical analysis on networking data (such as on the route reported in a traceroute, or the timing observations of a connection).

- A system for logging information, and reporting information to the user.

Event Diagram

It may be helpful to consider the preliminary design for how the system should process data received from the network. This is a particularly important aspect of the system, as it determines much of the overall architecture. To that end, figure 1 shows part of the graph of events. The graph is a directed graph with an edge from node A to node B representing an event that A could produce and B would receive.

Of course, there is no restriction on how many modules may subscribe to an event. Therefore, it is possible to easily add extra functionality, such as to log how many Ethernet frames are sent and received simply by allowing

another module (in this example, a log) to receive these events. Since this graph would be managed by the subscription service (described in more detail below), this diagram does not contain every node or edge that would be in the final system, in order to prevent the diagram becoming unwieldy.

The subscription mechanism is in charge of maintaining the edges in this graph so that the system can be constructed in a much more modular way than would otherwise be achievable. Modules subscribe to events they would like to receive using the subscription mechanism. This enables a new module to be added to the system without making changes to other parts of the system that would otherwise be necessary to link the new module in. The key here is to ensure this subscription mechanism is sufficiently flexible to accommodate an event model that provides tidy linking of loosely coupled modules. It becomes clear that a good implementation of this subscription mechanism is important.

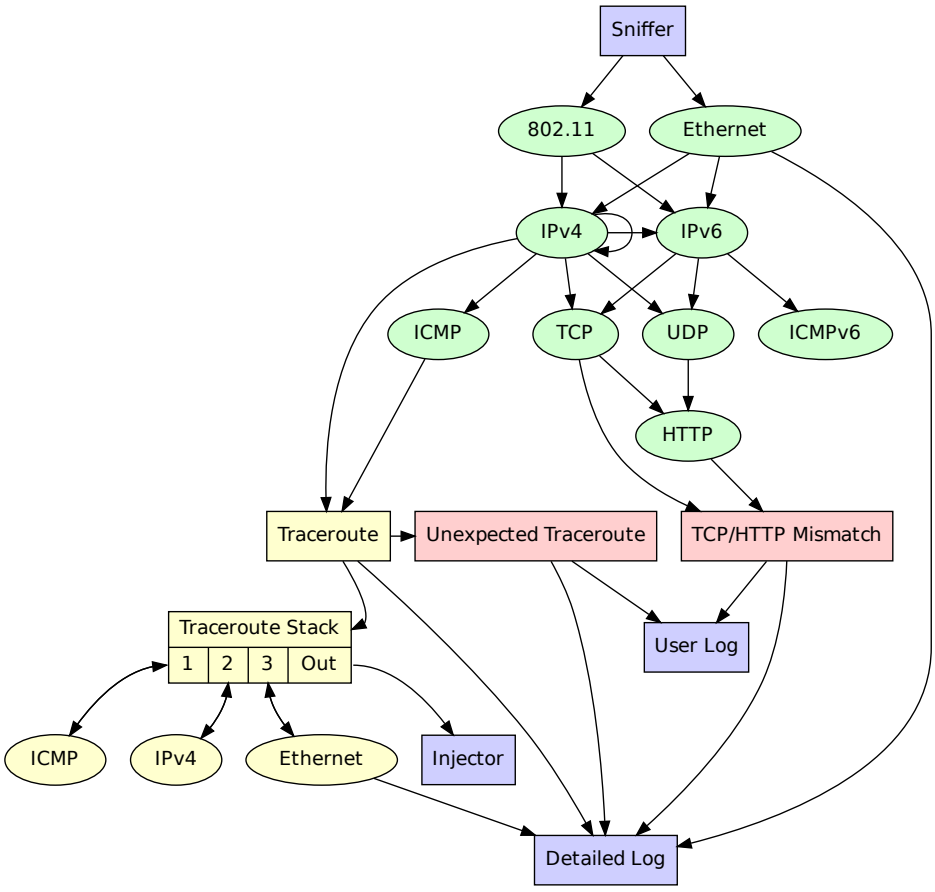

Figure 1: Event Graph

The graph is coloured to make explanation of the different parts of the system easier to explain. The blue nodes are responsible for interacting with the outside world – for example writing to a log file, or listening to the network. The green nodes are protocols for recognising received data. The yellow nodes are responsible for interacting with the network – in this example, performing traceroutes. The red nodes detect anomalies[8].

As an example, if a UDP in IP in Ethernet packet is received, the sniffer would produce an Ethernet frame received event. Since the Ethernet protocol subscribes to Ethernet frame received events, this would be processed by the Ethernet protocol, which would then produce an IP packet received event (as well as an Ethernet type 0x0800 packet received event). The IP protocol (which subscribes to these events) would process this packet, and produce a UDP packet received event which will be processed by the UDP protocol.

The Traceroute Stack warrants explanation. It is set up by the Traceroute and is responsible for coordinating the generation and receipt the events necessary to send a complete packet over the network.

In this example, the Ethernet receiving and sending nodes both log information using the event system. In the developed system, many of the other nodes might write to the log too – particularly for debugging purposes. This graph may contain cycles to permit the processing of encapsulated packets – for example, to permit IPv4-in-IPv4.

Extensions

This project presents several different possible extensions beyond the core functionality necessary to satisfy the success criteria. These include:

- The most obvious extension is the inclusion of additional detectable anomalies.
- Discovering information about non-anomalous operation of the network, such as building a map of the network. This could use information already gathered for anomaly detection. Continuing the example, traceroutes performed to detect anomalous routing could also be used to build this map.
- Providing defences against some malicious anomalies. For example, a port scan might be hindered by pretending to have open ports that are in fact closed.

[8] Note: I have included "TCP/HTTP Mismatch" because it is a good example of how an anomaly may interact with more than one protocol. I would need to do further research before I could determine how feasible this anomaly is to detect.

Success Criteria

This project could be evaluated against several different criteria. These might include:

- The system should be able to detect a number of different anomalies, using active and passive techniques where appropriate. These anomalies might include:
  - Unusual path revealed by traceroute
  - Unusual time taken to reach a hop
  - Port scanning of the local machine
  - The receipt of unusual packet types
- An analysis of the memory and CPU requirements of the system, and how this is affected by the amount and type of network traffic received. It is desirable that the system can cope with a typical, but high load.
- An analysis of the relationship between the traffic received, and the amount and type of traffic generated. It is desirable that the system should not generate unacceptable traffic under any circumstance.
- The system should be implemented in a modular way that would enable further functionality to be added easily.

Evaluation Method

These success criteria (as well as others that are discovered to be useful) will be important in the evaluation section of the dissertation, and to establish that the produced software works.

The system should correctly detect the implemented anomalies. This can be established qualitatively – the system either tends to make the correct judgement or not. In many cases, it may also be possible to perform a quantitative analysis by simulating many random instances of an anomaly, and establishing the system's false positive and negative rate (being aware that these figures may not be representative of the real world if the artificial nature of the tested anomaly instances poorly reflects reality). Evaluation of this category of criteria will include these types of testing. A good working implementation of the project will be indicated by an ability to detect anomalies with a low false positive and negative rate, except where one would reasonably expect this to be difficult. Further, functionality such as logging should be demonstrated to work reliably.

It is desirable that the system should be implemented in a modular way. This will be evaluated by considering the work that would have to be done to extend the system – to recognise additional anomalies, or to introduce functionality that is quite different from that which has been implemented. I will also consider how dependent the modules are on each other, and how well abstractions that are designed to provide loose coupling work in practice. This category of criteria, although important to this project, warrants a more qualitative evaluation.

I will also perform an analysis of the computing resources required by the project. This will be a very quantitative analysis that will provide an indication of how practical it would be to operate the system in a range of different environments. This includes an analysis and consideration of the traffic generated by the system, as well as processing time, memory, and other resources required. This section is particularly amenable to a more numerical analysis – for example, producing graphs such as memory CPU time usage caused by traffic received. Some of this analysis can also be automated by producing a test framework that can collect the data for these graphs. Success for this category of criteria will require that the system should work well on modest end user hardware under typical load. Ideally, however, we would like that the system can provide most of its functionality in much more challenging environments. In this case, we may consider the project to be especially successful by these criteria.

Timetable and Milestones

Work Packet 1

1st October 2011—31st October 2011

- Complete paper work, for example:
  - This project proposal
  - Resource request paperwork
- Set up framework for the project. For example:
  - GIT repository
  - Dissertation template
  - Log (planned to be a directory in the git repository)
- Research and study documentation for example:

  - RFCs
  - Software documentation

- Investigate a rough set of tools, techniques and ideas that will be required, for example:
  - Virtual machine software (and software to run in the VM)
  - Available libraries and their APIs
  - Other software and techniques that already exist (for example, network centred IDSs)

Work Packet 2

31st October 2011—21st November 2011

- Continue with work already started
- Produce a design for the project
- Begin documenting the design. This design might include information about:
  - An overall architecture
  - Protocols
  - Anomalies
  - Reporting and logging
  - Configuration
  - The linking between protocols, anomalies, and other reporting and logging
  - Consideration of data structures and algorithms, and how they will be used
- Set up the framework for the code base
- Build a very simple packet dissection system based on the above:
  - A small number of protocols (mainly packet based)
  - A small number of demo anomalies (not necessarily real ones)
  - Simple reporting (possibly just producing information about received packets)

  - The subscription mechanism should be mostly implemented at this stage. The system should be implemented in a modular way from the outset.

- Refine the design

Work Packet 3

21st November 2011—9th December 2011

- Extend the code built in the previous work packet. This might include:
  - Basic TCP support
  - Basic HTTP support
  - A few simple anomalies
  - Begin writing the configuration mechanism.
- Begin implementing framework and tools for testing the project. This might include:
  - A network of virtual machines
  - Anomaly simulation tools
- Write the dissertation introduction to a good standard
- Write the dissertation preparation to a partial standard

Work Packet 4

9th December 2011—23rd December 2011

- Refine testing framework and tools
- Begin early work on the dissertation implementation
- Roughly and informally evaluate the work done so far. Consider what should be changed or redesigned. Produce and experiment with new ideas. Much of this work might be recorded as notes in the log.

Work Packet 5

23rd December 2011—16th January 2012

- Rework code to refine it, and include ideas considered in the last work packet. This is largely (though not exclusively) a "stop and polish" process. The code should be in the following state by the end of this item:
  - Core framework stuff should be quite solid by this point
  - Good progress should be made implementing the testing framework. The testing system should, by this point, be able to measure many of the criteria set out in the success criteria. All of the implemented anomalies should be tested by the system. The testing process should be able to establish the CPU and memory requirements of the system under relatively normal conditions.
- Write the dissertation preparation to a good standard
- Continue working on the dissertation implementation
- Informally begin to consider in more detail how the evaluation might proceed
- Start limited empirical evaluation
- Begin writing the progress report

Work Packet 6

16th January 2012—2nd February 2012

- Rework and polish the testing framework as necessary. By this stage, much of the quantitative analysis should be implemented, and preliminary data should be available.
- Begin to expand the feature set. This may be a fairly limited process in this work packet in order to ensure sufficient time is devoted to the progress report, and to making the code stable. Work that should begin might include:
  - Implement more anomalies, and experiment with more ambitious anomalies
  - Implement the associated testing

- Reach a state suitable for a progress report - this is a "stop and polish" step
- Write the progress report

Work Packet 7

2nd February 2012—20th February 2012

- Heavy work on expanding feature set. By the end of this process, all the planned features of the project should be implemented. At this stage, extensions may also have been implemented if the project is going particularly well.
- Further empirical testing. By this time, the testing framework should be able to produce data for the full evaluation of all the success criteria.
- Partial dissertation implementation
- Begin to write the evaluation
- Informal ideas to prepare to refine above development

Work Packet 8

20th February 2012—12th March 2012

- Polish code where necessary
- Gather most of the remaining empirical testing data
- Have the dissertation implementation written to a good standard
- Have the dissertation evaluation written to a partial standard
- Begin writing the rest of the dissertation

Work Packet 9

12th March 2012—24th March 2012

- Polish most of the dissertation
- Finish writing the end of the dissertation
- Polish this too

Work Packet 10

24^{th} March 2012—30^{th} April 2012

- Fix anything that needs fixing